Scientific Annals of Computer Science vol. 36 (1), 2026, pp. 1–26
doi: 10.47743/SACS.2026.1.1

On the Formalization of the Notion of a Concurrent Algorithm

C.A. Middelburg [1]

Abstract

Previous papers give accounts of quests for satisfactory formalizations of the classical informal notion of an algorithm and the contemporary informal notion of an interactive algoritm. In this paper, an attempt is made to generalize the results of the former quest to the contemporary informal notion of a concurrent algorithm. The notion of a concurrent proto-algorithm is introduced. The thought is that concurrent algorithms are equivalence classes of concurrent proto-algorithms under an appropriate equivalence relation. As in the previous papers referred to above, three equivalence relations are defined. Two of them are deemed to be bounds for an appropriate equivalence relation and the third is likely an appropriate one. The connection between concurrency and non-determinism in the presented setting is also addressed.

Keywords: concurrent algorithm, concurrent proto-algorithm, algorithmic equivalence, computational equivalence, non-determinism

1 Introduction

In [9] is reported on a quest for a satisfactory formalization of the classical informal notion of an algorithm. By this is meant the notion of an algorithm that is informally characterized in many works from the mathematical and computer science literature, including standard works such as [5, 6, 8, 12]. In the works concerned, an algorithm is informally characterized by properties

This work is licensed under the Creative Commons Attribution Licence (CC BY)

[1]Informatics Institute, Faculty of Science, University of Amsterdam, Science Park 900, 1098 XH Amsterdam, the Netherlands, Email: C.A.Middelburg@uva.nl

that are considered the most important ones of an algorithm. The notion of a proto-algorithm defined in [9] captures virtually all properties that are mentioned in those works. Moreover, a proto-algorithm expresses a pattern of behaviour to solve all instances of a computational problem without using a machine model or an algorithmic language. This makes it neither too abstract nor too concrete to be a suitable basis for investigating what exactly an algorithm is in the setting of the kinds of computation that are based on a model of parallel computation.

In this paper, the algorithms of the kind that emerged due to the advent of parallel computation are called concurrent algorithms. They can be briefly described as follows: "The pattern of behaviour expressed by a concurrent algorithm consists of multiple parts that can take place concurrently". Algorithms that are called parallel algorithms in the computer science literature are usually concurrent algorithms satisfying restrictions imposed by some model of parallel computation (cf. [7]). The notion of a concurrent algorithm is intended to encompass all types of parallel algorithms that can be found in the computer science literature. Using the term concurrent algorithm avoids confusion with the existing uses of the term parallel algorithm.

Despite the fact that parallel algorithms of various types are widespread today, no work can be found in the computer science literature that aims at a satisfactory formalization of one of the informal notions of a parallel algorithm that exist today. This motivated me to start a quest for a satisfactory formalization of the notion of a concurrent algorithm. An important goal of this quest is gaining more insight into what the different types of parallel algorithms have in common and how they all differ from classical algorithms.

Due to the advent of parallel computation, several related models of parallel computation have been proposed. Some of them are based on a particular variant of Turing machines, such as the parallel Turing machine model of computation proposed in [14]. However, most are based on a variant of a RAM (Random Access Machine). They include synchronous parallel RAM models of computation, such as the models proposed in [2, 3, 13], and asynchronous parallel RAM models of computation, such as the models proposed in [1, 7, 11]. In this paper, the starting point for the formalization of the notion of a concurrent algorithm is a characterization of the notion by properties suggested by those models of parallel computation.

Based on that characterization, the notion of a concurrent proto-

algorithm is introduced. Like the notion of a proto-algorithm, the notion of a concurrent proto-algorithm is introduced with the thought that concurrent algorithms are equivalence classes of concurrent proto-algorithms under an appropriate equivalence relation. Two equivalence relations are defined that give bounds for an appropriate equivalence relation and one equivalence relation is defined that is likely an appropriate one. The connection between concurrency and non-determinism in the setting of concurrent proto-algorithms and the last equivalence relation is also addressed.

This paper is organized as follows. First, the basic notions and notations used in this paper are introduced (Section 2). Next, an intuitive characterization of the notion of a concurrent algorithm is given by properties that are considered to belong to the most important ones of a concurrent algorithm (Section 3). After that, the formal notion of a concurrent proto-algorithm is introduced and the isomorphism relation on concurrent proto-algorithms is defined (Section 4). Then, it is defined what a run of a concurrent proto-algorithm is and what is computed by a concurrent proto-algorithm (Section 5). Thereafter, two equivalence relations on concurrent proto-algorithms are defined (Section 6). Following that, the connection between non-determinism and concurrency in the presented setting is addressed (Section 7). Finally, some concluding remarks are made (Section 8).

In [10], an attempt is made to generalize the work on the classical notion of an algorithm presented in [9] to the notion of an interactive algorithm. In the current paper, an attempt is made to generalize the work on the classical notion of an algorithm presented in [9] to the notion of a concurrent algorithm. This has led to text overlap of the current paper with [9] and [10], especially in Sections 2 and 3. To easily recognize similarities and differences, the presentation styles of the current paper and [10] are quite similar.

2 Preliminaries

In this section, the basic notions and notations used in this paper are introduced.

The notion of a concurrent proto-algorithm will be formally defined in Section 4 in terms of three auxiliary notions. The definition of one of these auxiliary notions is based on the well-known notion of a rooted labeled directed graph. However, the definitions of this notion given in the mathematical and computer science literature vary. Therefore, the definition that is used in this paper is given.

Definition 1 *A* rooted labeled directed graph *G is a sextuple $(V, E, L_\mathrm{v}, L_\mathrm{e}, l, r)$, where:*

- *V is a non-empty finite set, whose members are called the* vertices *of G;*
- *E is a subset of $V \times V$, whose members are called the* edges *of G;*
- *L_v is a countable set, whose members are called the* vertex labels *of G;*
- *L_e is a countable set, whose members are called the* edge labels *of G;*
- *l is a partial function from $V \cup E$ to $L_\mathrm{v} \cup L_\mathrm{e}$ such that*

  $$\begin{aligned}&\text{for all } v \in V \text{ for which } l(v) \text{ is defined, } l(v) \in L_\mathrm{v} \text{ and}\\ &\text{for all } e \in E \text{ for which } l(e) \text{ is defined, } l(e) \in L_\mathrm{e},\end{aligned}$$

  called the labeling function *of G;*
- *$r \in V$, called the* root *of G.*

The additional graph theoretical notions defined below are also used.

Definition 2 *Let $G = (V, E, L_\mathrm{v}, L_\mathrm{e}, l, r)$ be a rooted labeled directed graph. Then a* cycle *in G is a sequence $v_1 \ldots v_{n+1} \in V^*$ such that, for all $i \in \{1, \ldots, n\}$, $(v_i, v_{i+1}) \in E$, $\mathrm{card}(\{v_1, \ldots, v_n\}) = n$, and $v_1 = v_{n+1}$. Let also $v \in V$. Then the* indegree *of v, written* $\mathrm{indeg}(v)$*, is* $\mathrm{card}(\{v' \mid (v', v) \in E\})$ *and the* outdegree *of v, written* $\mathrm{outdeg}(v)$*, is* $\mathrm{card}(\{v' \mid (v, v') \in E\})$.

For convenience, the following notions concerning tuples are used:

Definition 3 *Let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be sets, let $a_1 \in \mathcal{A}_1$, …, $a_n \in \mathcal{A}_n$, let $\boldsymbol{a} = (a_1, \ldots, a_n)$, and let $i \in \{1, \ldots, n\}$. Then the* ith element of *$\boldsymbol{a}$, written $\boldsymbol{a}(i)$, is a_i. Let also $a \in \mathcal{A}_i$. Then $\boldsymbol{a}$* with the *i*th element replaced by *a, written $\boldsymbol{a}[i \mapsto a]$, is $(a_1, \ldots, a_{i-1}, a, a_{i+1}, \ldots, a_n)$.*

The following notations may not be entirely standard:

- we write $\mathbb{N}^+$ for the set $\{n \in \mathbb{N} \mid n > 0\}$ of positive natural numbers;
- we write $\mathcal{A}^n$, where $\mathcal{A}$ is a set and $n \in \mathbb{N}^+$, for the n-fold Cartesian product of $\mathcal{A}$ with itself;
- we write $\mathcal{P}(\mathcal{A})$, where $\mathcal{A}$ is a set, for the set of all subsets of $\mathcal{A}$.

In Section 4, we assume the existence of a dummy value $\bot$ that is not a member of certain sets. The following special notations concerning $\bot$ are used:

- we write $\mathcal{A}_\bot$, where $\mathcal{A}$ is a set such that $\bot \notin \mathcal{A}$, for $\mathcal{A} \cup \{\bot\}$;
- we write $\bot^n$, where $n \in \mathbb{N}^+$, for the unique member of the set $\{\bot\}^n$.

In Section 5, we consider multi-valued functions, i.e. functions from a set $\mathcal{A}$ to the set of all subsets of a set $\mathcal{A}'$. The following notion concerning multi-valued functions is used:

Definition 4 *Let $\mathcal{A}$ and $\mathcal{A}'$ be sets, and let f and g be functions from $\mathcal{A}$ to $\mathcal{P}(\mathcal{A}')$. Then f is* smaller then *g if $f(s) \subset g(s)$ for all $s \in \mathcal{A}$.*

3 The Informal Notion of a Concurrent Algorithm

In this paper, the notion of a classical algorithm concerns the original notion of an algorithm, i.e. the notion that is intuitively characterized in standard works from the mathematical and computer science literature such as [5, 6, 8, 12], and the notion of a concurrent algorithm concerns a generalization of the original notion of an algorithm. Seeing the terminology used for proposed generalizations of the original notion of an algorithm, a classical algorithm could also be called a deterministic sequential non-interactive algorithm.

The characterizations of the informal notion of a classical algorithm referred to above indicate that a classical algorithm is considered to express a pattern of behaviour by which all instances of a computational problem can be solved. This calls for a description of a classical computational problem that does not refer to the notion of a classical algorithm:

> A classical computational problem is a problem where, given a value from a certain set of input values, a value from a certain set of output values that is in a certain relation to the given input value must be produced if it exists. The input values are also called the instances of the problem and an output value that is in the certain relation to a given input value is also called a solution for the instance concerned.

From the existing viewpoints on what a classical algorithm is, it follows that the following properties must be considered the most important ones of a classical algorithm:

1. a classical algorithm is a finite expression of a pattern of behaviour by which all instances of a classical computational problem can be solved;

2. the pattern of behaviour expressed by a classical algorithm is made up of discrete steps, each of which consists of performing an elementary operation or inspecting an elementary condition unless it is the initial step or a final step;

3. the pattern of behaviour expressed by a classical algorithm is such that there is one possible step immediately following a step that consists of performing an operation;

4. the pattern of behaviour expressed by a classical algorithm is such that there is one possible step immediately following a step that consists of inspecting a condition for each outcome of the inspection;

5. the pattern of behaviour expressed by a classical algorithm is such that the initial step consists of inputting an input value of the problem concerned;

6. the pattern of behaviour expressed by a classical algorithm is such that, for each input value of the problem concerned for which a correct output value exists, a final step is reached after a finite number of steps and that final step consists of outputting a correct output value for that input value;

7. the steps involved in the pattern of behaviour expressed by a classical algorithm are precisely and unambiguously defined and can be performed exactly in a finite amount of time.

The notion of a classical algorithm covers algorithms that express a pattern of sequential behaviour. It does not cover algorithms in which parts can take place concurrently. The notion of a concurrent algorithm covers algorithms that express a pattern of concurrent behaviour. A concurrent algorithm can be roughly characterized as consisting of a number of components that are classical algorithms except that:

- there is only one component whose initial step consists of inputting an input value, there is only one component whose final steps consist of outputting an output value, and those components are the same;

- some elementary operations that can be performed by a component involve two values: a value available only to the component concerned and a value available to all components.

The above characterization has been derived from the viewpoints on concurrent computation that are expressed in computer science publications such as [1, 2, 3, 7]. It reflects a rather operational view of what a concurrent algorithm is. In a more abstract view of what a concurrent algorithm is, a concurrent algorithm expresses a collection of patterns of concurrent behaviour that are equivalent in some well-defined way. We will come back to this at the end of Section 4.

4 Concurrent Proto-Algorithms

In this section, the notion of a concurrent proto-algorithm is introduced. The thought is that concurrent algorithms are equivalence classes of concurrent proto-algorithms under an appropriate equivalence relation. An equivalence relation that is likely an appropriate one is introduced in Section 6.

We proceed with defining the three auxiliary notions in terms of which the notion of a concurrent proto-algorithm will be defined, starting with the notion of an alphabet. This notion concerns the symbols used to refer to the operations and conditions involved in the steps of which the pattern of behaviour expressed by a concurrent algorithm is made up.

Definition 5 *An* alphabet Σ *is a quadruple* $(F, F_{\mathrm{s}}, F_{\mathrm{g}}, P)$*, where:*

- F *is a countable set of* processing function symbols *of* Σ*;*
- F_{s} *is a countable set of* setting function symbols *of* Σ*;*
- F_{g} *is a countable set of* getting function symbols *of* Σ*;*
- P *is a countable set of* predicate symbols *of* Σ*;*
- F*,* F_{s}*,* F_{g}*, and* P *are disjoint sets and* $\mathsf{ini}, \mathsf{fin} \in F$*.*

Σ *is called a* classical *alphabet if* $F_{\mathrm{s}} = \emptyset$ *and* $F_{\mathrm{g}} = \emptyset$*.*

We write $\widehat{F}$ and $\widetilde{F}$, where F is the set of processing function symbols of an alphabet, for the sets $F \setminus \{\mathsf{fin}\}$ and $F \setminus \{\mathsf{ini}, \mathsf{fin}\}$, respectively.

The function symbols and predicate symbols of an alphabet refer to the operations and conditions, respectively, involved in the steps of which the pattern of behaviour expressed by a concurrent algorithm is made up. The function symbols ini and fin refer to inputting a first input value and outputting a last output value, respectively.

We are now ready to define the notions of a concurrent-Σ-algorithm component graph and a Σ-interpretation. They concern the pattern of behaviour expressed by a concurrent algorithm.

Definition 6 *Let $\Sigma = (F, F_\mathrm{s}, F_\mathrm{g}, P)$ be an alphabet. Then a* concurrent-Σ-algorithm component graph *G is a rooted labeled directed graph $(V, E, L_\mathrm{v}, L_\mathrm{e}, l, r)$ such that*

- $L_\mathrm{v} = F \cup F_\mathrm{s} \cup F_\mathrm{g} \cup P$*;*
- $L_\mathrm{e} = \{0, 1\}$*;*
- *for all $v \in V$:*
  - $\mathrm{indeg}(v) = 0$ *iff* $v = r$*;*
  - *if* $l(v) = \mathsf{ini}$*, then* $\mathrm{indeg}(v) = 0$*;*
  - *if* $l(v) = \mathsf{fin}$*, then* $\mathrm{outdeg}(v) = 0$*;*
  - *if* $l(v) \in \widehat{F} \cup F_\mathrm{s} \cup F_\mathrm{g}$*, then* $\mathrm{outdeg}(v) = 1$ *and, for the unique* $v' \in V$ *such that* $(v, v') \in E$*,* $l((v, v'))$ *is undefined;*
  - *if* $l(v) \in P$*, then* $\mathrm{outdeg}(v) = 2$ *and, for the unique* $v', v'' \in V$ *such that* $v' \neq v''$ *and* $(v, v'), (v, v'') \in E$*, both* $l((v, v'))$ *and* $l((v, v''))$ *are defined and* $l((v, v')) \neq l((v, v''))$*;*
- $l(r) = \mathsf{ini}$ *iff there exists a* $v \in V$ *such that* $l(v) = \mathsf{fin}$*;*
- *for all cycles* $v_1 \ldots v_{n+1}$ *in* G*, there exists a* $v \in \{v_1, \ldots, v_n\}$ *such that* $l(v) \in \widehat{F} \cup F_\mathrm{s} \cup F_\mathrm{g}$*.*

G is called a concurrent-Σ-algorithm main *component graph if* $l(r) = \mathsf{ini}$*.*

In the above definition, the condition on cycles in a concurrent-Σ-algorithm component graph excludes infinitely many consecutive steps, each of which consists of inspecting a condition.

In the above definition, the conditions regarding the vertices and edges of a concurrent-Σ-algorithm component graph correspond to the essential

properties of the components of a concurrent algorithm described in Section 3 that concern their structure. Adding an interpretation of the symbols of the alphabet Σ to a concurrent-Σ-algorithm component graph yields something that has all of the essential properties of the components of a concurrent algorithm described in Section 3.

Definition 7 *Let $\Sigma = (F, F_{\mathrm{s}}, F_{\mathrm{g}}, P)$ be an alphabet. Then a* Σ-interpretation *$\mathcal{I}$ is a quadruple $(D, D_{\mathrm{i}}, D_{\mathrm{o}}, I)$, where:*

- *D is a set, called the* algorithm domain *of $\mathcal{I}$;*
- *D_{i} is a finitely generated set, called the* input domain *of $\mathcal{I}$;*
- *D_{o} is a finitely generated set, called the* output domain *of $\mathcal{I}$;*
- *I is a total function from $F \cup F_{\mathrm{s}} \cup F_{\mathrm{g}} \cup P$ to the set of all total computable functions from D_{i} to D, D to D_{o}, D to D, $D \times D$ to D or D to $\{0,1\}$ such that:*
  - *$I(\mathsf{ini})$ is a function from D_{i} to D;*
  - *$I(\mathsf{fin})$ is a function from D to D_{o};*
  - *for all $f \in \widetilde{F}$, $I(f)$ is a function from D to D;*
  - *for all $f \in F_{\mathrm{s}} \cup F_{\mathrm{g}}$, $I(f)$ is a function from $D \times D$ to D;*
  - *for all $p \in P$, $I(p)$ is a function from D to $\{0,1\}$;*
- *there does not exist a $D' \subset D$ such that:*
  - *for all $d_{\mathrm{i}} \in D_{\mathrm{i}}$, $I(\mathsf{ini})(d_{\mathrm{i}}) \in D'$;*
  - *for all $f \in \widetilde{F}$, for all $d \in D'$, $I(f)(d) \in D'$;*
  - *for all $f \in F_{\mathrm{s}} \cup F_{\mathrm{g}}$, for all $d, d' \in D'$, $I(f)(d, d') \in D'$;*
- *for all $d, d' \in D$:*
  - *for all $f \in F_{\mathrm{s}}$, $f(d, f(d, d')) = f(d, d')$ and there exists a $f' \in F_{\mathrm{g}}$ such that $f'(d, f(d, d')) = d$;*
  - *for all $f \in F_{\mathrm{g}}$, $f(f(d, d'), d') = f(d, d')$ and there exists a $f' \in F_{\mathrm{s}}$ such that $f'(f(d, d'), d') = d'$.*

The finite generation condition on D_i and D_o and the minimality condition on D in the above definition is considered desirable. They guarantee that all elements of D_i, D_o, and D have a finite representation, which is generally expected of the values involved in the steps of an algorithm. However, these conditions have not yet been shown to be essential in establishing results. The condition on F_s and F_g in the above definition is considered characteristic of the nature of the setting functions and matching getting functions, but has not yet been shown to be essential in establishing results.

The pattern of behavior expressed by a concurrent algorithm can be fully represented by the combination of an alphabet Σ, a number of concurrent-Σ-algorithm component graphs $G_1, \ldots, G_n$, and a Σ-interpretation $\mathcal{I}$. This brings us to defining the notion of a concurrent proto-algorithm.

Definition 8 *A* concurrent proto-algorithm *A is a triple $(\Sigma, \boldsymbol{G}, \mathcal{I})$, where:*

- *Σ is an alphabet, called the* alphabet *of A;*
- *$\boldsymbol{G}$ is a tuple of concurrent-Σ-algorithm component graphs, called the* algorithm component graphs *of A, of which exactly one is a concurrent-Σ-algorithm main component graph;*
- *$\mathcal{I}$ is a Σ-interpretation, called the* interpretation *of A.*

In [9], a definition of the notion of a classical proto-algorithm is given.[1] A corollary of that definition and the definition of a concurrent proto-algorithm given above is that a classical proto-algorithm is a special case of a concurrent proto-algorithm.

Corollary 9 *A concurrent proto-algorithm $(\Sigma, \boldsymbol{G}, \mathcal{I})$ is a classical proto-algorithm (as defined in [9]) if Σ is a classical alphabet and $\boldsymbol{G}$ is a tuple of one concurrent-Σ-algorithm component graph.*

Many definitions and results concerning concurrent proto-algorithms from the current paper are generalizations of definitions and results from [9].

Henceforth, we assume a dummy value $\bot$ such that, for all concurrent proto-algorithms $A = (\Sigma, \boldsymbol{G}, \mathcal{I})$, where $\Sigma = (F, F_\mathrm{s}, F_\mathrm{g}, P)$, $\boldsymbol{G} = (G_1, \ldots, G_n)$ with $G_i = (V_i, E_i, L_{\mathrm{v}i}, L_{\mathrm{e}i}, l_i, r_i)$ $(1 \leq i \leq n)$, and $\mathcal{I} = (D, D_\mathrm{i}, D_\mathrm{o}, I)$, $\bot \notin V_i$ for each $i \in \{1, \ldots, n\}$, $\bot \notin D$, $\bot \notin D_\mathrm{i}$, $\bot \notin D_\mathrm{o}$, and $\bot \notin \{1, \ldots, n\}$.

The intuition is that a concurrent proto-algorithm is something that goes from one state to another.

[1] Classical proto-algorithms are simply called proto-algorithms in [9].

Definition 10 *Let $A = (\Sigma, \boldsymbol{G}, \mathcal{I})$ be a concurrent proto-algorithm, where $\Sigma = (F, F_\mathrm{s}, F_\mathrm{g}, P)$, $\boldsymbol{G} = (G_1, \ldots, G_n)$ with $G_i = (V_i, E_i, L_{\mathrm{v}i}, L_{\mathrm{e}i}, l_i, r_i)$ $(1 \leq i \leq n)$, and $\mathcal{I} = (D, D_\mathrm{i}, D_\mathrm{o}, I)$. Then a* state *of A is a triple $(d_\mathrm{i}, c, d_\mathrm{o})$, where $d_\mathrm{i} \in D_{\mathrm{i}\bot}$, $c \in (V_{1\bot} \times \ldots \times V_{n\bot}) \times {D_\bot}^n \times D_\bot \times \{1, \ldots, n\}_\bot$, and $d_\mathrm{o} \in D_{\mathrm{o}\bot}$, such that:*

- *if $c = (\bot^n, \bot^n, \bot, \bot)$, then $d_\mathrm{i} = \bot$ iff $d_\mathrm{o} \neq \bot$;*
- *if $c \neq (\bot^n, \bot^n, \bot, \bot)$, then $d_\mathrm{i} = \bot$ and $d_\mathrm{o} = \bot$.*

A state $(d_\mathrm{i}, c, d_\mathrm{o})$ of A is called an initial state *of A if $d_\mathrm{i} \neq \bot$, a state $(d_\mathrm{i}, c, d_\mathrm{o})$ of A is called a* final state *of A if $d_\mathrm{o} \neq \bot$, and a state $(d_\mathrm{i}, c, d_\mathrm{o})$ of A is called an* internal state *of A if $d_\mathrm{i} = \bot$ and $d_\mathrm{o} = \bot$.*

Henceforth, we write $\mathcal{S}_A$, where A is a concurrent proto-algorithm, for the set of all states of A. We also write $\mathcal{S}_A^{\mathrm{ini}}$ for the set of all initial states of A, $\mathcal{S}_A^{\mathrm{fin}}$ for the set of all final states of A, and $\mathcal{S}_A^{\mathrm{int}}$ for the set of all internal states of A. Moreover, we write $\bot_\mathrm{c}$ for the tuple $(\bot^n, \bot^n, \bot, \bot)$.

An internal state $(\bot, ((v_1, \ldots, v_n), (d_1, \ldots, d_n), d, j), \bot)$ of a proto-algorithm A can be largely explained from the point of view of state machines:

- v_i is the control state of the ith component of A $(1 \leq i \leq n)$;
- d_i is the private data state of the ith component of A $(1 \leq i \leq n)$;
- d is the shared data state of the components of A.

Moreover, j indicates that this state is one in which a step to a next state is made by the jth component of A.

Suppose that $A = (\Sigma, \boldsymbol{G}, \mathcal{I})$ is a concurrent proto-algorithm, where $\Sigma = (F, F_\mathrm{s}, F_\mathrm{g}, P)$, $\boldsymbol{G} = (G_1, \ldots, G_n)$ with $G_i = (V_i, E_i, L_{\mathrm{v}i}, L_{\mathrm{e}i}, l_i, r_i)$ $(1 \leq i \leq n)$, $\mathcal{I} = (D, D_\mathrm{i}, D_\mathrm{o}, I)$, and $\boldsymbol{r} = (r_1, \ldots, r_n)$. A goes from one state to the next state by making a step, it starts in an initial state, and if it does not keep making steps forever, it stops in a final state. The following is an informal explanation of how the state that A is in determines what the possible steps to a next state consists of and to what next states they lead:

- if A is in initial state $(d_\mathrm{i}, \bot_\mathrm{c}, \bot)$ and $i \in \{1, \ldots, n\}$ is such that $l(\boldsymbol{r}(i)) = \mathsf{ini}$, then, for each $j \in \{1, \ldots, n\}$ with $\mathrm{outdeg}(\boldsymbol{r}(j)) > 0$, a step to a next state is possible that consists of applying function $I(\mathsf{ini})$ to d_i and leads to the unique internal state $(\bot, (\boldsymbol{r}[i \mapsto v'], \bot^n[i \mapsto d'], \bot, j), \bot)$ such that $(\boldsymbol{r}(i), v') \in E_i$ and $I(\mathsf{ini})(d_\mathrm{i}) = d'$;

- if A is in internal state $(\bot, (\boldsymbol{v}, \boldsymbol{d}, d, i), \bot)$ and $l(\boldsymbol{v}(i)) \in \widetilde{F}$, then, for each $j \in \{1, \ldots, n\}$ with $\mathrm{outdeg}(\boldsymbol{v}(j)) > 0$, a step to a next state is possible that consists of applying function $I(l(\boldsymbol{v}(i)))$ to $\boldsymbol{d}(i)$ and leads to the unique internal state $(\bot, (\boldsymbol{v}[i \mapsto v'], \boldsymbol{d}[i \mapsto d'], d, j), \bot)$ such that $(\boldsymbol{v}(i), v') \in E_i$, and $I(l(\boldsymbol{v}(i)))(\boldsymbol{d}(i)) = d'$;

- if A is in internal state $(\bot, (\boldsymbol{v}, \boldsymbol{d}, d, i), \bot)$ and $l(\boldsymbol{v}(i)) \in F_\mathrm{s}$, then, for each $j \in \{1, \ldots, n\}$ with $\mathrm{outdeg}(\boldsymbol{v}(j)) > 0$, a step to a next state is possible that consists of applying function $I(l(\boldsymbol{v}(i)))$ to $(\boldsymbol{d}(i), d)$ and leads to the unique internal state $(\bot, (\boldsymbol{v}[i \mapsto v'], \boldsymbol{d}, d', j), \bot)$ such that $(\boldsymbol{v}(i), v') \in E_i$, and $I(l(\boldsymbol{v}(i)))(\boldsymbol{d}(i), d) = d'$;

- if A is in internal state $(\bot, (\boldsymbol{v}, \boldsymbol{d}, d, i), \bot)$ and $l(\boldsymbol{v}(i)) \in F_\mathrm{g}$, then, for each $j \in \{1, \ldots, n\}$ with $\mathrm{outdeg}(\boldsymbol{v}(j)) > 0$, a step to a next state is possible that consists of applying function $I(l(\boldsymbol{v}(i)))$ to $(\boldsymbol{d}(i), d)$ and leads to the unique internal state $(\bot, (\boldsymbol{v}[i \mapsto v'], \boldsymbol{d}[i \mapsto d'], d, j), \bot)$ such that $(\boldsymbol{v}(i), v') \in E_i$, and $I(l(\boldsymbol{v}(i)))(\boldsymbol{d}(i), d) = d'$;

- if A is in internal state $(\bot, (\boldsymbol{v}, \boldsymbol{d}, d, i), \bot)$ and $l(\boldsymbol{v}(i)) \in P$, then a step to a next state is possible that consists of applying function $I(l(\boldsymbol{v}(i)))$ to $\boldsymbol{d}(i)$ and leads to the unique internal state $(\bot, (\boldsymbol{v}[i \mapsto v'], \boldsymbol{d}, d, i), \bot)$ such that $(\boldsymbol{v}(i), v') \in E_i$, and $I(l(\boldsymbol{v}(i)))(\boldsymbol{d}(i)) = l((\boldsymbol{v}(i), v'))$;

- if A is in internal state $(\bot, (\boldsymbol{v}, \boldsymbol{d}, d, i), \bot)$, $l(\boldsymbol{v}(i)) = \mathsf{fin}$, and for all $j \in \{1, \ldots, n\}$, $\mathrm{outdeg}(\boldsymbol{v}(j)) = 0$, then a step to a next state is possible that consists of applying function $I(\mathsf{fin})$ to $\boldsymbol{d}(i)$ and leads to the unique final state $(\bot, \bot_\mathrm{c}, d_\mathrm{o})$ such that $I(\mathsf{fin})(\boldsymbol{d}(i)) = d_\mathrm{o}$.

This informal explanation of how the state that A is in determines what the possible next states are, is formalized by the algorithmic step function δ^a_A defined in Section 5.

Because concurrent proto-algorithms are considered too concrete to be called concurrent algorithms, the term concurrent proto-algorithm has been chosen instead of the term concurrent algorithm. For example, from a mathematical point of view, it is natural to consider the behavioral patterns expressed by isomorphic concurrent proto-algorithms to be the same.

Although it is intuitive clear what isomorphism of concurrent proto-algorithms is, its precise definition is long and tedious.

Definition 11 *Let $A = (\Sigma, \boldsymbol{G}, \mathcal{I})$ and $A' = (\Sigma', \boldsymbol{G}', \mathcal{I}')$ be concurrent proto-algorithms, where $\Sigma = (F, F_\mathrm{s}, F_\mathrm{g}, P)$, $\Sigma' = (F', F'_\mathrm{s}, F'_\mathrm{g}, P')$, $\boldsymbol{G} = (G_1, \ldots, G_n)$*

with $G_i = (V_i, E_i, L_{\mathrm{v}i}, L_{\mathrm{e}i}, l_i, r_i)$ $(1 \leq i \leq n)$, $\boldsymbol{G}' = (G'_1, \ldots, G'_{n'})$ *with* $G'_j = (V'_j, E'_j, L_{\mathrm{v}}{}'_j, L_{\mathrm{e}}{}'_j, l'_j, r'_j)$ $(1 \leq j \leq n')$, $\mathcal{I} = (D, D_{\mathrm{i}}, D_{\mathrm{o}}, I)$, *and* $\mathcal{I}' = (D', D_{\mathrm{i}}{}', D_{\mathrm{o}}{}', I')$. *Then* A *and* A' *are* isomorphic, *written* $A \cong A'$, *if there exist bijections* $\beta_{\mathrm{i}} : \{1, \ldots, n\} \to \{1, \ldots, n\}$, $\beta_{\mathrm{f}} : F \to F'$, $\beta_{\mathrm{f_s}} : F_{\mathrm{s}} \to F'_{\mathrm{s}}$, $\beta_{\mathrm{f_g}} : F_{\mathrm{g}} \to F'_{\mathrm{g}}$, $\beta_{\mathrm{p}} : P \to P'$, $\beta_{\mathrm{v}i} : V_i \to V'_{\beta_{\mathrm{i}}(i)}$ *for* $i \in \{1, \ldots, n\}$, $\beta_{\mathrm{d}} : D \to D'$, $\beta_{\mathrm{d_i}} : D_{\mathrm{i}} \to D_{\mathrm{i}}{}'$, $\beta_{\mathrm{d_o}} : D_{\mathrm{o}} \to D_{\mathrm{o}}{}'$, *and* $\beta_{\mathrm{b}} : \{0,1\} \to \{0,1\}$ *such that:*

- $n = n'$;
- *for all* $i \in \{1, \ldots, n\}$:
  - *for all* $v, v' \in V_i$, $(v, v') \in E_i$ *iff* $(\beta_{\mathrm{v}i}(v), \beta_{\mathrm{v}i}(v')) \in E'_{\beta_{\mathrm{i}}(i)}$;
  - *for all* $v \in V_i$ *with* $l_i(v) \in F$, $\beta_{\mathrm{f}}(l_i(v)) = l'_{\beta_{\mathrm{i}}(i)}(\beta_{\mathrm{v}i}(v))$;
  - *for all* $v \in V_i$ *with* $l_i(v) \in F_{\mathrm{s}}$, $\beta_{\mathrm{f_s}}(l_i(v)) = l'_{\beta_{\mathrm{i}}(i)}(\beta_{\mathrm{v}i}(v))$;
  - *for all* $v \in V_i$ *with* $l_i(v) \in F_{\mathrm{g}}$, $\beta_{\mathrm{f_g}}(l_i(v)) = l'_{\beta_{\mathrm{i}}(i)}(\beta_{\mathrm{v}i}(v))$;
  - *for all* $v \in V_i$ *with* $l_i(v) \in P$, $\beta_{\mathrm{p}}(l_i(v)) = l'_{\beta_{\mathrm{i}}(i)}(\beta_{\mathrm{v}i}(v))$;
  - *for all* $(v, v') \in E_i$ *with* $l_i((v, v'))$ *defined,* $\beta_{\mathrm{b}}(l_i((v, v'))) = l'_{\beta_{\mathrm{i}}(i)}((\beta_{\mathrm{v}i}(v), \beta_{\mathrm{v}i}(v')))$;
- $\beta_{\mathrm{f}}(\mathsf{ini}) = \mathsf{ini}$ *and* $\beta_{\mathrm{f}}(\mathsf{fin}) = \mathsf{fin}$;
- *for all* $d_{\mathrm{i}} \in D_{\mathrm{i}}$, $\beta_{\mathrm{d}}(I(\mathsf{ini})(d_{\mathrm{i}})) = I'(\mathsf{ini})(\beta_{\mathrm{d_i}}(d_{\mathrm{i}}))$;
- *for all* $d \in D$, $\beta_{\mathrm{d_o}}(I(\mathsf{fin})(d)) = I'(\mathsf{fin})(\beta_{\mathrm{d}}(d))$;
- *for all* $d \in D$ *and* $f \in \widetilde{F}$, $\beta_{\mathrm{d}}(I(f)(d)) = I'(\beta_{\mathrm{f}}(f))(\beta_{\mathrm{d}}(d))$;
- *for all* $d, d' \in D$ *and* $f \in F_{\mathrm{s}}$, $\beta_{\mathrm{d}}(I(f)(d, d')) = I'(\beta_{\mathrm{f_s}}(f))(\beta_{\mathrm{d}}(d), \beta_{\mathrm{d}}(d'))$;
- *for all* $d, d' \in D$ *and* $f \in F_{\mathrm{g}}$, $\beta_{\mathrm{d}}(I(f)(d, d')) = I'(\beta_{\mathrm{f_g}}(f))(\beta_{\mathrm{d}}(d), \beta_{\mathrm{d}}(d'))$;
- *for all* $d \in D$ *and* $p \in P$, $\beta_{\mathrm{b}}(I(p)(d)) = I'(\beta_{\mathrm{p}}(p))(\beta_{\mathrm{d}}(d))$.

Concurrent proto-algorithms may also be considered too concrete in a way not covered by isomorphism of concurrent proto-algorithms. This issue is addressed in Section 6 and leads there to the introduction of two other equivalence relations.

A concurrent proto-algorithm could also be defined as a quadruple $(D, D_{\mathrm{i}}, D_{\mathrm{o}}, \overline{\boldsymbol{G}})$ where $\overline{\boldsymbol{G}}$ is a tuple of graphs that differ from concurrent-Σ-algorithm component graphs in that their vertex labels are computable

functions from D_i to D, D to D_o, D to D, $D \times D$ to D or D to $\{0,1\}$ instead of function and predicate symbols from Σ. I consider the definition of a concurrent proto-algorithm given earlier more insightful because it isolates as much as possible the operations to be performed and the conditions to be inspected from the structure of a concurrent proto-algorithm.

5 On Steps, Runs and What is Computed

In Section 4, the intuition was given that a concurrent proto-algorithm A is something that goes through states. It was informally explained how the state that it is in determines what the possible steps to a next state consists of and to what next states they lead. The algorithmic step function δ^a_A that is defined below formalizes this. The computational step function δ^c_A that is also defined below is like the algorithmic step function δ^a_A, but conceals the steps that consist of inspecting conditions.

Definition 12 *Let $A = (\Sigma, \boldsymbol{G}, \mathcal{I})$ be a concurrent proto-algorithm, where $\Sigma = (F, F_\mathrm{s}, F_\mathrm{g}, P)$, $\boldsymbol{G} = (G_1, \ldots, G_n)$ with $G_i = (V_i, E_i, L_{\mathrm{v}i}, L_{\mathrm{e}i}, l_i, r_i)$ $(1 \leq i \leq n)$, and $\mathcal{I} = (D, D_\mathrm{i}, D_\mathrm{o}, I)$, and let $\boldsymbol{r} = (r_1, \ldots, r_n)$. Then the* algorithmic step function δ^a_A induced by A *is the smallest total function from $\mathcal{S}_A$ to $\mathcal{P}(\mathcal{S}_A) \setminus \emptyset$ such that for all $d, d' \in D$, $d_\mathrm{i} \in D_\mathrm{i}$, $d_\mathrm{o} \in D_\mathrm{o}$, $v'_1 \in V_1$, $\ldots$, $v'_n \in V_n$, $\boldsymbol{d} \in D^n$, $\boldsymbol{v} \in V_1 \times \ldots \times V_n$, and $i, j \in \{1, \ldots, n\}$:*[2]

$$\begin{aligned}
&\delta^\mathrm{a}_A((d_\mathrm{i}, \bot_\mathrm{c}, \bot)) \ni (\bot, (\boldsymbol{r}[i \mapsto v'_i], \bot^n[i \mapsto d'], \bot, j), \bot)\\
&\quad \textit{if } l(\boldsymbol{r}(i)) = \mathsf{ini},\ (\boldsymbol{r}(i), v'_i) \in E_i,\ I(\mathsf{ini})(d_\mathrm{i}) = d',\ \textit{and } \mathrm{outdeg}(\boldsymbol{r}(j)) > 0,\\
\delta^\mathrm{a}_A((\bot, (\boldsymbol{v}, \boldsymbol{d}, d, i), \bot)) &\ni (\bot, (\boldsymbol{v}[i \mapsto v'_i], \boldsymbol{d}[i \mapsto d'], d, j), \bot)\\
&\quad \textit{if } l(\boldsymbol{v}(i)) \in \widetilde{F},\ (\boldsymbol{v}(i), v'_i) \in E_i,\ I(l(\boldsymbol{v}(i)))(\boldsymbol{d}(i)) = d',\ \textit{and}\\
&\qquad \mathrm{outdeg}(\boldsymbol{v}(j)) > 0,\\
\delta^\mathrm{a}_A((\bot, (\boldsymbol{v}, \boldsymbol{d}, d, i), \bot)) &\ni (\bot, (\boldsymbol{v}[i \mapsto v'_i], \boldsymbol{d}, d', j), \bot)\\
&\quad \textit{if } l(\boldsymbol{v}(i)) \in F_\mathrm{s},\ (\boldsymbol{v}(i), v'_i) \in E_i,\ I(l(\boldsymbol{v}(i)))(\boldsymbol{d}(i), d) = d',\ \textit{and}\\
&\qquad \mathrm{outdeg}(\boldsymbol{v}(j)) > 0,\\
\delta^\mathrm{a}_A((\bot, (\boldsymbol{v}, \boldsymbol{d}, d, i), \bot)) &\ni (\bot, (\boldsymbol{v}[i \mapsto v'_i], \boldsymbol{d}[i \mapsto d'], d, j), \bot)\\
&\quad \textit{if } l(\boldsymbol{v}(i)) \in F_\mathrm{g},\ (\boldsymbol{v}(i), v'_i) \in E_i,\ I(l(\boldsymbol{v}(i)))(\boldsymbol{d}(i), d) = d',\ \textit{and}\\
&\qquad \mathrm{outdeg}(\boldsymbol{v}(j)) > 0,\\
\delta^\mathrm{a}_A((\bot, (\boldsymbol{v}, \boldsymbol{d}, d, i), \bot)) &\ni (\bot, (\boldsymbol{v}[i \mapsto v'_i], \boldsymbol{d}, d, i), \bot)\\
&\quad \textit{if } l(\boldsymbol{v}(i)) \in P,\ (\boldsymbol{v}(i), v'_i) \in E_i,\ \textit{and } I(l(\boldsymbol{v}(i)))(\boldsymbol{d}(i)) = l((\boldsymbol{v}(i), v'_i)),
\end{aligned}$$

[2] In this definition, we write $S \ni s$ instead of $s \in S$.

$\delta^{\mathrm{a}}_A((\bot,(\boldsymbol{v},\boldsymbol{d},d,i),\bot)) \ni (\bot,\bot_{\mathrm{c}},d_{\mathrm{o}})$
if $l(\boldsymbol{v}(i)) = \mathsf{fin}$, $I(\mathsf{fin})(\boldsymbol{d}(i)) = d_{\mathrm{o}}$, *and*
for all $k \in \{1,\ldots,n\}$, $\mathrm{outdeg}(\boldsymbol{v}(k)) = 0$,
$\delta^{\mathrm{a}}_A((\bot,\bot_{\mathrm{c}},d_{\mathrm{o}})) \ni (\bot,\bot_{\mathrm{c}},d_{\mathrm{o}})$

and the computational step function δ^{c}_A induced by A *is the smallest total function from* $\mathcal{S}_A$ *to* $\mathcal{P}(\mathcal{S}_A)\setminus\emptyset$ *such that, for all* $d,d' \in D$, $d_{\mathrm{i}} \in D_{\mathrm{i}}$, $d_{\mathrm{o}} \in D_{\mathrm{o}}$, $v'_1 \in V_1$, …, $v'_n \in V_n$, $\boldsymbol{d} \in D^n$, $\boldsymbol{v} \in V_1 \times \ldots \times V_n$, $i,j \in \{1,\ldots,n\}$, *and* $s \in \mathcal{S}_A$*:*

$\delta^{\mathrm{c}}_A((d_{\mathrm{i}},\bot_{\mathrm{c}},\bot)) \ni (\bot,(\boldsymbol{r}[i\mapsto v'_i],\bot^n[i\mapsto d'],\bot,j),\bot)$
if $l(\boldsymbol{r}(i)) = \mathsf{ini}$, $(\boldsymbol{r}(i),v'_i) \in E_i$, $I(\mathsf{ini})(d_{\mathrm{i}}) = d'$, *and* $\mathrm{outdeg}(\boldsymbol{r}(j)) > 0$,
$\delta^{\mathrm{c}}_A((\bot,(\boldsymbol{v},\boldsymbol{d},d,i),\bot)) \ni (\bot,(\boldsymbol{v}[i\mapsto v'_i],\boldsymbol{d}[i\mapsto d'],d,j),\bot)$
if $l(\boldsymbol{v}(i)) \in \widetilde{F}$, $(\boldsymbol{v}(i),v'_i) \in E_i$, $I(l(\boldsymbol{v}(i)))(\boldsymbol{d}(i)) = d'$, *and*
$\mathrm{outdeg}(\boldsymbol{v}(j)) > 0$,
$\delta^{\mathrm{c}}_A((\bot,(\boldsymbol{v},\boldsymbol{d},d,i),\bot)) \ni (\bot,(\boldsymbol{v}[i\mapsto v'_i],\boldsymbol{d},d',j),\bot)$
if $l(\boldsymbol{v}(i)) \in F_{\mathrm{s}}$, $(\boldsymbol{v}(i),v'_i) \in E_i$, $I(l(\boldsymbol{v}(i)))(\boldsymbol{d}(i),d) = d'$, *and*
$\mathrm{outdeg}(\boldsymbol{v}(j)) > 0$,
$\delta^{\mathrm{c}}_A((\bot,(\boldsymbol{v},\boldsymbol{d},d,i),\bot)) \ni (\bot,(\boldsymbol{v}[i\mapsto v'_i],\boldsymbol{d}[i\mapsto d'],d,j),\bot)$
if $l(\boldsymbol{v}(i)) \in F_{\mathrm{g}}$, $(\boldsymbol{v}(i),v'_i) \in E_i$, $I(l(\boldsymbol{v}(i)))(\boldsymbol{d}(i),d) = d'$, *and*
$\mathrm{outdeg}(\boldsymbol{v}(j)) > 0$,
$\delta^{\mathrm{c}}_A((\bot,(\boldsymbol{v},\boldsymbol{d},d,i),\bot)) \ni s$
if $l(\boldsymbol{v}(i)) \in P$, $(\boldsymbol{v}(i),v'_i) \in E_i$, $I(l(\boldsymbol{v}(i)))(\boldsymbol{d}(i)) = l((\boldsymbol{v}(i),v'_i))$, *and*
$\delta^{\mathrm{c}}_A((\bot,(\boldsymbol{v}[i\mapsto v'_i],\boldsymbol{d},d,i),\bot)) \ni s$,
$\delta^{\mathrm{c}}_A((\bot,(\boldsymbol{v},\boldsymbol{d},d,i),\bot)) \ni (\bot,\bot_{\mathrm{c}},d_{\mathrm{o}})$
if $l(\boldsymbol{v}(i)) = \mathsf{fin}$, $I(\mathsf{fin})(\boldsymbol{d}(i)) = d_{\mathrm{o}}$, *and*
for all $k \in \{1,\ldots,n\}$, $\mathrm{outdeg}(\boldsymbol{v}(k)) = 0$,
$\delta^{\mathrm{c}}_A((\bot,\bot_{\mathrm{c}},d_{\mathrm{o}})) \ni (\bot,\bot_{\mathrm{c}},d_{\mathrm{o}})$.

Below, we define what the possible runs of a concurrent proto-algorithm on an input value is and what is computed by a concurrent proto-algorithm. To this end, we first give some auxiliary definitions.

In the coming definitions, we write $\mathcal{A}^\infty$ for the set of all non-empty sequences over the set $\mathcal{A}$ that are finite or countably infinite, $\langle\rangle$ for the empty sequence, $\langle a\rangle$ for the sequence having a as sole element, $\alpha \frown \alpha'$ for the concatenation of the sequences α and α', and $|\alpha|$ for the length of the sequence α.

The sets of sequences of states that A goes through from a given state according to the algorithmic step function and computational step function are given by the functions srs^{a}_A and srs^{c}_A, respectively, defined below.

Definition 13 *Let $A = (\Sigma, \boldsymbol{G}, \mathcal{I})$ be a concurrent proto-algorithm. Then the* algorithmic semi-run set function srs^{a}_A induced by A *and the* computational semi-run set function srs^{c}_A induced by A *are the total functions from $\mathcal{S}_A$ to $\mathcal{P}({\mathcal{S}_A}^\infty)$ such that for all $s \in \mathcal{S}_A$:*

$$\begin{aligned} srs^{\mathrm{a}}_A(s) &= \{\langle s\rangle \frown \sigma \mid (\exists s' \in \delta^{\mathrm{a}}_A(s))\ \sigma \in srs^{\mathrm{a}}_A(s')\} && \text{if } s \notin \mathcal{S}^{\mathrm{fin}}_A,\\ srs^{\mathrm{a}}_A(s) &= \{\langle s\rangle\} && \text{if } s \in \mathcal{S}^{\mathrm{fin}}_A. \end{aligned}$$

and

$$\begin{aligned} srs^{\mathrm{c}}_A(s) &= \{\langle s\rangle \frown \sigma \mid (\exists s' \in \delta^{\mathrm{c}}_A(s))\ \sigma \in srs^{\mathrm{c}}_A(s')\} && \text{if } s \notin \mathcal{S}^{\mathrm{fin}}_A,\\ srs^{\mathrm{c}}_A(s) &= \{\langle s\rangle\} && \text{if } s \in \mathcal{S}^{\mathrm{fin}}_A. \end{aligned}$$

A $\sigma \in {\mathcal{S}_A}^\infty$ is called an algorithmic semi-run *of A if there exists an $s \in \mathcal{S}_A$ such that $\sigma \in srs^{\mathrm{a}}_A(s)$.*

Henceforth, we write $\mathcal{SR}^{\mathrm{a}}_A$, where A is a concurrent proto-algorithm, for the set of all algorithmic semi-runs of A.

When defining what is computed by a concurrent proto-algorithm, a distinction must be made between convergent semi-runs and divergent semi-runs.

Definition 14 *Let $A = (\Sigma, \boldsymbol{G}, \mathcal{I})$ be a concurrent proto-algorithm, and let $\sigma \in \mathcal{SR}^{\mathrm{a}}_A$. Then σ* is divergent *if there exists a suffix σ' of σ such that $\sigma' \in {\mathcal{S}^{\mathrm{int}}_A}^\infty$, and σ* is convergent *if σ is not divergent.*

The output value eventually produced by a given convergent algorithmic semi-run is given by the function ov_A defined below.

Definition 15 *Let $A = (\Sigma, G, \mathcal{I})$ be a concurrent proto-algorithm. Then the* output value extraction function ov_A for A *is the total function from $\{\sigma \in \mathcal{SR}^{\mathrm{a}}_A \mid \sigma \text{ is convergent}\}$ to D_{o} such that for all $(d_{\mathrm{i}}, c, d_{\mathrm{o}}) \in \mathcal{S}_A$ and $\sigma \in \mathcal{S}^\infty_A \cup \{\langle\rangle\}$ such that $\langle(d_{\mathrm{i}}, c, d_{\mathrm{o}})\rangle \frown \sigma \in \mathcal{SR}^{\mathrm{a}}_A$ and $\langle(d_{\mathrm{i}}, c, d_{\mathrm{o}})\rangle \frown \sigma$ is convergent:*

$$\begin{aligned} ov_A(\langle(d_{\mathrm{i}}, c, d_{\mathrm{o}})\rangle \frown \sigma) &= d_{\mathrm{o}} && \text{if } d_{\mathrm{o}} \neq \bot,\\ ov_A(\langle(d_{\mathrm{i}}, c, d_{\mathrm{o}})\rangle \frown \sigma) &= ov_A(\sigma) && \text{if } d_{\mathrm{o}} = \bot. \end{aligned}$$

A run of a concurrent proto-algorithm is a semi-run of the concurrent proto-algorithm concerned that starts in an initial state. The following definition concerns what the possible runs of a non-interactive proto-algorithm on an input value are. As with steps, a distinction is made between algorithmic runs and computational runs.

Definition 16 *Let $A = (\Sigma, \boldsymbol{G}, \mathcal{I})$ be a concurrent proto-algorithm, where $\mathcal{I} = (D, D_\mathrm{i}, D_\mathrm{o}, I)$, and let $d_\mathrm{i} \in D_\mathrm{i}$. Then the* algorithmic run set of A on d_i*, written $\rho^\mathrm{a}_A(d_\mathrm{i})$, is $srs^\mathrm{a}_A((d_\mathrm{i}, \bot_\mathrm{c}, \bot))$ and the* computational run set of A on d_i*, written $\rho^\mathrm{c}_A(d_\mathrm{i})$, is $srs^\mathrm{c}_A((d_\mathrm{i}, \bot_\mathrm{c}, \bot))$.*

The following definition concerns what is computed by a concurrent proto-algorithm.

Definition 17 *Let $A = (\Sigma, \boldsymbol{G}, \mathcal{I})$ be a concurrent proto-algorithm, where $\mathcal{I} = (D, D_\mathrm{i}, D_\mathrm{o}, I)$. Then the* relation $\widehat{A}$ computed by A *is the relation from D_i to D_o such that for all $d_\mathrm{i} \in D_\mathrm{i}$ and $d_\mathrm{o} \in D_\mathrm{o}$:*

$(d_\mathrm{i}, d_\mathrm{o}) \in \widehat{A}$ iff there exists a convergent $\sigma \in \rho^\mathrm{a}_A(d_\mathrm{i})$ such that $d_\mathrm{o} = ov_A(\sigma)$.

If $A = (\Sigma, \boldsymbol{G}, \mathcal{I})$ is a concurrent proto-algorithm where $\boldsymbol{G}$ is a tuple of only one concurrent-Σ-algorithm component graph, then the relation $\widehat{A}$ computed by A is functional, i.e. $\widehat{A}$ is (the graph of) a partial function.

6 Algorithmic and Computational Equivalence

If a concurrent proto-algorithm A' can mimic a concurrent proto-algorithm A step-by-step, then we say that A is algorithmically simulated by A'. If the steps that consist of inspecting conditions are ignored, then we say that A is computationally simulated by A'. Algorithmic and computational simulation can be formally defined using the step functions defined in Section 5.

Definition 18 *Let $A = (\Sigma, \boldsymbol{G}, \mathcal{I})$ and $A' = (\Sigma', \boldsymbol{G}', \mathcal{I}')$ be concurrent proto-algorithms. Then an* algorithmic simulation of A by A' *is a set $R \subseteq \mathcal{S}_A \times \mathcal{S}_{A'}$ such that:*

- *for all $s \in \mathcal{S}_A$ and $s' \in \mathcal{S}_{A'}$:*
  - *if $s \in \mathcal{S}^\mathrm{ini}_A$, then there exists an $s' \in \mathcal{S}^\mathrm{ini}_{A'}$ such that $(s, s') \in R$;*
  - *if $s' \in \mathcal{S}^\mathrm{fin}_{A'}$, then there exists an $s \in \mathcal{S}^\mathrm{fin}_A$ such that $(s, s') \in R$;*
  - *if $(s, s') \in R$ and $t \in \delta^\mathrm{a}_A(s)$, then there exists a $t' \in \delta^\mathrm{a}_{A'}(s')$ such that $(t, t') \in R$;*
- *for all $(s, s') \in R$:*
  - *$s \in \mathcal{S}^\mathrm{ini}_A$ iff $s' = \mathcal{S}^\mathrm{ini}_{A'}$;*
  - *$s \in \mathcal{S}^\mathrm{fin}_A$ iff $s' = \mathcal{S}^\mathrm{fin}_{A'}$*

and a computational simulation of A by A' *is a set* $R \subseteq \mathcal{S}_A \times \mathcal{S}_{A'}$ *such that:*

- *for all* $s \in \mathcal{S}_A$ *and* $s' \in \mathcal{S}_{A'}$*:*
  - *if* $s \in \mathcal{S}_A^{\mathrm{ini}}$*, then there exists an* $s' \in \mathcal{S}_{A'}^{\mathrm{ini}}$ *such that* $(s, s') \in R$*;*
  - *if* $s' \in \mathcal{S}_{A'}^{\mathrm{fin}}$*, then there exists an* $s \in \mathcal{S}_A^{\mathrm{fin}}$ *such that* $(s, s') \in R$*;*
  - *if* $(s, s') \in R$ *and* $t \in \delta_A^{\mathrm{c}}(s)$*, then there exists a* $t' \in \delta_{A'}^{\mathrm{c}}(s')$ *such that* $(t, t') \in R$*;*
- *for all* $(s, s') \in R$*:*
  - $s \in \mathcal{S}_A^{\mathrm{ini}}$ *iff* $s' = \mathcal{S}_{A'}^{\mathrm{ini}}$*;*
  - $s \in \mathcal{S}_A^{\mathrm{fin}}$ *iff* $s' = \mathcal{S}_{A'}^{\mathrm{fin}}$*.*

A is algorithmically simulated by A', *written* $A \sqsubseteq_{\mathrm{a}} A'$, *if there exists an algorithmic simulation* R *of* A *by* A'*.*
A is computationally simulated by A', *written* $A \sqsubseteq_{\mathrm{c}} A'$, *if there exists a computational simulation* R *of* A *by* A'*.*
A is algorithmically equivalent to A', *written* $A \equiv_{\mathrm{a}} A'$, *if there exist an algorithmic simulation* R *of* A *by* A' *and an algorithmic simulation* R' *of* A' *by* A *such that* $R' = R^{-1}$*.*
A is computationally equivalent to A', *written* $A \equiv_{\mathrm{c}} A'$, *if there exist an computational simulation* R *of* A *by* A' *and an computational simulation* R' *of* A' *by* A *such that* $R' = R^{-1}$*.*

The conditions imposed on an algorithmic or computational simulation R of a concurrent proto-algorithm A by a concurrent proto-algorithm A' include, in addition to the usual transfer conditions, also conditions that guarantee that a state of A is only related by R to a state of A' of the same kind (initial, final or internal). Lemma 19 and Theorem 20 given below would not hold without the additional conditions.

There may be states of a concurrent proto-algorithm in which there is a choice from multiple possible steps to a next state. The condition $R' = R^{-1}$ imposed on simulations R and R' witnessing (algorithmic or computational) equivalence of concurrent proto-algorithms A and A' guarantees that A and A' have the same choice structure.

The next lemma will be used in the proof of the theorem that follows it. In this lemma and the proof of the theorem that follows it, we write $\alpha[n]$ for the nth element of the sequence α if there exists a prefix of α with length n and otherwise the last element of α.

Lemma 19 *Let $A = (\Sigma, \boldsymbol{G}, \mathcal{I})$ and $A' = (\Sigma', \boldsymbol{G}', \mathcal{I}')$ be concurrent proto-algorithms, where $\Sigma = (F, F_{\mathrm{s}}, F_{\mathrm{g}}, P)$, $\Sigma' = (F', F_{\mathrm{s}}', F_{\mathrm{g}}', P')$, $\boldsymbol{G} = (G_1, \ldots, G_n)$ with $G_i = (V_i, E_i, L_{\mathrm{v}i}, L_{\mathrm{e}i}, l_i, r_i)$ $(1 \leq i \leq n)$, $\boldsymbol{G}' = (G_1', \ldots, G_{n'}')$ with $G_j' = (V_j', E_j', L_{\mathrm{v}}{}'_j, L_{\mathrm{e}}{}'_j, l_j', r_j')$ $(1 \leq j \leq n')$, $\mathcal{I} = (D, D_{\mathrm{i}}, D_{\mathrm{o}}, I)$, and $\mathcal{I}' = (D', D_{\mathrm{i}}{}', D_{\mathrm{o}}{}', I')$, and let $R \subseteq \mathcal{S}_A \times \mathcal{S}_{A'}$ and $\gamma_{\mathrm{i}} : D_{\mathrm{i}} \to D_{\mathrm{i}}{}'$ be such that, for all $d_{\mathrm{i}} \in D_{\mathrm{i}}$, $((d_{\mathrm{i}}, \bot_{\mathrm{c}}, \bot), (\gamma_{\mathrm{i}}(d_{\mathrm{i}}), \bot_{\mathrm{c}}, \bot)) \in R$. Then R is an algorithmic simulation of A by A' only if, for all $d_{\mathrm{i}} \in D_{\mathrm{i}}$, for all $\sigma \in \rho_A^{\mathrm{a}}(d_{\mathrm{i}})$, there exists a $\sigma' \in \rho_{A'}^{\mathrm{a}}(\gamma_{\mathrm{i}}(d_{\mathrm{i}}))$ such that, for all $n \in \mathbb{N}^+$, $(\sigma[n], \sigma'[n]) \in R$.*

Proof: The proof is the same as the proof of Lemma 19 in [10]. □

The following theorem tells us that, if a concurrent proto-algorithm A is algorithmically simulated by a concurrent proto-algorithm A', then (a) the relation computed by A' models the relation computed by A (in the sense of e.g. [4]) and (b) for each convergent algorithmic run of A, the simulation results in a convergent algorithmic run of A' consisting of the same number of algorithmic steps.

Theorem 20 *Let $A = (\Sigma, \boldsymbol{G}, \mathcal{I})$ and $A' = (\Sigma', \boldsymbol{G}', \mathcal{I}')$ be concurrent proto-algorithms, where $\mathcal{I} = (D, D_{\mathrm{i}}, D_{\mathrm{o}}, I)$, and $\mathcal{I}' = (D', D_{\mathrm{i}}{}', D_{\mathrm{o}}{}', I')$. Then $A \sqsubseteq_{\mathrm{a}} A'$ only if there exist total functions $\gamma_{\mathrm{i}} : D_{\mathrm{i}} \to D_{\mathrm{i}}{}'$ and $\gamma_{\mathrm{o}} : D_{\mathrm{o}}{}' \to D_{\mathrm{o}}$ such that:*

(1) for all $d_{\mathrm{i}} \in D_{\mathrm{i}}$ and $d_{\mathrm{o}} \in D_{\mathrm{o}}$, $(d_{\mathrm{i}}, d_{\mathrm{o}}) \in \widehat{A}$ only if there exists a $d_{\mathrm{o}}' \in D_{\mathrm{o}}{}'$ such that $(\gamma_{\mathrm{i}}(d_{\mathrm{i}}), d_{\mathrm{o}}') \in \widehat{A'}$ and $\gamma_{\mathrm{o}}(d_{\mathrm{o}}') = d_{\mathrm{o}}$;

(2) for all $d_{\mathrm{i}} \in D_{\mathrm{i}}$, for all convergent $\sigma \in \rho_A^{\mathrm{a}}(d_{\mathrm{i}})$, there exists a convergent $\sigma' \in \rho_{A'}^{\mathrm{a}}(\gamma_{\mathrm{i}}(d_{\mathrm{i}}))$ with $\gamma_{\mathrm{o}}(ov_{A'}(\sigma')) = ov_A(\sigma)$ such that $|\sigma| = |\sigma'|$.

Proof: The proof is the same as the proof of Theorem 20 in [10]. □

It is easy to see that Theorem 20 goes through as far as (1) and (2) are concerned if algorithmic simulation is replaced by computational simulation. However, (2) does not go through if algorithmic simulation is replaced by computational simulation.

The following theorem tells us how isomorphism, algorithmic equivalence, and computational equivalence are related.

Theorem 21 *Let A and A' be concurrent proto-algorithms. Then:*

(1) $A \cong A'$ *only if* $A \equiv_{\mathrm{a}} A'$ *(2)* $A \equiv_{\mathrm{a}} A'$ *only if* $A \equiv_{\mathrm{c}} A'$.

Proof: Let $A = (\Sigma, \boldsymbol{G}, \mathcal{I})$ and $A' = (\Sigma', \boldsymbol{G}', \mathcal{I}')$ be concurrent proto-algorithms, where $\Sigma = (F, F_\mathrm{s}, F_\mathrm{g}, P)$, $\Sigma' = (F', F'_\mathrm{s}, F'_\mathrm{g}, P')$, $\boldsymbol{G} = (G_1, \ldots, G_n)$ with $G_i = (V_i, E_i, L_{\mathrm{v}i}, L_{\mathrm{e}i}, l_i, r_i)$ $(1 \leq i \leq n)$, $\boldsymbol{G}' = (G'_1, \ldots, G'_{n'})$ with $G'_j = (V'_j, E'_j, L_{\mathrm{v}j}', L_{\mathrm{e}j}', l'_j, r'_j)$ $(1 \leq j \leq n')$, $\mathcal{I} = (D, D_\mathrm{i}, D_\mathrm{o}, I)$, and $\mathcal{I}' = (D', D_\mathrm{i}', D_\mathrm{o}', I')$.

Part 1. Because $A \cong A'$, there exist bijections β_i, $\beta_{\mathrm{v}1}$, …, $\beta_{\mathrm{v}n}$, β_d, $\beta_{\mathrm{d_i}}$, and $\beta_{\mathrm{d_o}}$ as in the definition of $\cong$. Let β_i, $\beta_{\mathrm{v}1}$, …, $\beta_{\mathrm{v}n}$, β_d, $\beta_{\mathrm{d_i}}$, and $\beta_{\mathrm{d_o}}$ be bijections as in the definition of $\cong$, let β^*_i, $\beta^*_{\mathrm{v}1}$, …, $\beta^*_{\mathrm{v}n}$, β^*_d, $\beta^*_{\mathrm{d_i}}$, and $\beta^*_{\mathrm{d_o}}$ be the extensions of β_i, $\beta_{\mathrm{v}1}$, …, $\beta_{\mathrm{v}n}$, β_d, $\beta_{\mathrm{d_i}}$, and $\beta_{\mathrm{d_o}}$, respectively, with the dummy value $\bot$ such that $\beta^*_\mathrm{i}(\bot) = \bot$, $\beta^*_{\mathrm{v}1}(\bot) = \bot$, …, $\beta^*_{\mathrm{v}n}(\bot) = \bot$, $\beta^*_\mathrm{d}(\bot) = \bot$, $\beta^*_{\mathrm{d_i}}(\bot) = \bot$, and $\beta^*_{\mathrm{d_o}}(\bot) = \bot$, and let β be the bijection from $\mathcal{S}_A$ to $\mathcal{S}_{A'}$ defined by $\beta(d_\mathrm{i}, ((v_1, \ldots, v_n), (d_1, \ldots, d_n), d, i), d_\mathrm{o}) = (\beta^*_{\mathrm{d_i}}(d_\mathrm{i}), ((\beta^*_{\mathrm{v}1}(v_1), \ldots, \beta^*_{\mathrm{v}n}(v_n)), (\beta^*_\mathrm{d}(d_1), \ldots, \beta^*_\mathrm{d}(d_n)), \beta^*_\mathrm{d}(d), \beta^*_\mathrm{i}(i)), \beta^*_{\mathrm{d_o}}(d_\mathrm{o}))$. Moreover, let $R = \{(s, \beta(s)) \mid s \in \mathcal{S}_A\}$ and let $R' = \{(s, \beta^{-1}(s)) \mid s \in \mathcal{S}_{A'}\}$. Then R is an algorithmic simulation of A by A' and R' is an algorithmic simulation of A' by A. This is easily proved by showing that the conditions from the definition of an algorithmic simulation are satisfied for all $(s, s') \in R$ and for all $(s, s') \in R'$, respectively. Moreover, it follows immediately from the definitions of R and R' that $R' = R^{-1}$. Hence, $A \equiv_\mathrm{a} A'$.

Part 2. Because $A \equiv_\mathrm{a} A'$, there exists an algorithmic simulation of A by A' such that its inverse is an algorithmic simulation of A' by A. Let R be an algorithmic simulation of A by A' such that R^{-1} is an algorithmic simulation of A' by A. Then R is also a computational simulation of A by A' and R^{-1} is also an computational simulation of A' by A. This is easily proved by showing that the conditions from the definition of a computational simulation are satisfied for all $(s, s') \in R$ and for all $(s, s') \in R'$, respectively. Hence, $A \equiv_\mathrm{c} A'$. □

The opposite implications do not hold in general. That is, there exist concurrent proto-algorithms A and A' for which it does not hold that $A \cong A'$ if $A \equiv_\mathrm{a} A'$ and there exist concurrent proto-algorithms A and A' for which it does not hold that $A \equiv_\mathrm{a} A'$ if $A \equiv_\mathrm{c} A'$. In both cases, the construction of a general illustrating example can be obtained from the construction of a general illustrating example for classical proto-algorithms described in [9] by applying that construction to a component of a concurrent proto-algorithm.

The definition of algorithmic equivalence suggests that it is reasonable to consider the patterns of behaviour expressed by algorithmically equivalent concurrent proto-algorithms the same. This suggests in turn that concurrent algorithms can be considered equivalence classes of concurrent

proto-algorithms under algorithmic equivalence. The definition of computational equivalence does not suggest that it is reasonable to consider the patterns of behaviour expressed by computationally equivalent concurrent proto-algorithms the same because steps that consist of inspecting a condition are treated as if they do not belong to the patterns of behaviour. The relevance of the computational equivalence relation is that any equivalence relation that captures the sameness of the patterns of behaviour expressed by concurrent proto-algorithms to a higher degree than the algorithmic equivalence relation must be finer than the computational equivalence relation.

7 Concurrency versus Non-determinism in Algorithms

Until now, only concurrent proto-algorithms with deterministic components have been considered. In this section, concurrent proto-algorithms with non-deterministic components are also considered because there is an interesting connection between concurrency and non-determinism in the setting of proto-algorithms.

In order to define the notion of a concurrent proto-algorithm with non-deterministic components, we have to weaken, in the definition of the notion of a concurrent-Σ-algorithm component graph, the outdegree of vertices labeled with a function symbol other than fin to greater than or equal to 1.

Definition 22 *Let $\Sigma = (F, F_{\mathrm{s}}, F_{\mathrm{g}}, P)$ be an alphabet. A* non-deterministic concurrent-Σ-algorithm component graph *G is a rooted labeled directed graph $(V, E, L_{\mathrm{v}}, L_{\mathrm{e}}, l, r)$ that satisfies the same conditions as a concurrent-Σ-algorithm component graph except that the condition*

> *if $l(v) \in \widehat{F} \cup F_{\mathrm{s}} \cup F_{\mathrm{g}}$, then $\mathrm{outdeg}(v) = 1$ and, for the unique $v' \in V$ such that $(v, v') \in E$, $l((v, v'))$ is undefined*

is replaced by the condition

> *if $l(v) \in \widehat{F} \cup F_{\mathrm{s}} \cup F_{\mathrm{g}}$, then $\mathrm{outdeg}(v) \geq 1$ and, for each $v' \in V$ such that $(v, v') \in E$, $l((v, v'))$ is undefined.*

Now, the definition of the notion of a concurrent proto-algorithm with non-deterministic components is simple.

Definition 23 *A* concurrent proto-algorithm *A* with non-deterministic components *is a triple $(\Sigma, \boldsymbol{G}, \mathcal{I})$, where:*

- Σ *is an alphabet, called the* alphabet *of* A*;*
- $\boldsymbol{G}$ *is a tuple of non-deterministic concurrent-*Σ*-algorithm component graphs, called the* algorithm component graphs *of* A*;*
- $\mathcal{I}$ *is a* Σ*-interpretation, called the* interpretation *of* A*.*

It is easy to see that all definitions and results concerning concurrent proto-algorithms given in this paper, except Corollary 9, go through for concurrent proto-algorithms with non-deterministic components.

A special case of a concurrent proto-algorithm with non-deterministic components is a non-deterministic sequential proto-algorithm.

Definition 24 *A* non-deterministic sequential proto-algorithm *A is a concurrent proto-algorithm $(\Sigma, \boldsymbol{G}, \mathcal{I})$ with non-deterministic components, where Σ is a classical alphabet and $\boldsymbol{G}$ is a tuple of one non-deterministic concurrent-Σ-algorithm component graph.*

In the light of Corollary 9, the notion of a non-deterministic sequential proto-algorithm may be viewed as the non-deterministic variant of the notion of a classical proto-algorithm.

The following theorem concerns the connection between concurrent proto-algorithms and non-deterministic sequential proto-algorithms.

Theorem 25 *Let A be a concurrent proto-algorithm. Then there exists a non-deterministic sequential proto-algorithm A' such that $A \equiv_{\mathrm{a}} A'$.*

Proof: Let $A = (\Sigma, \boldsymbol{G}, \mathcal{I})$ be a concurrent proto-algorithm, where $\Sigma = (F, F_{\mathrm{s}}, F_{\mathrm{g}}, P)$, $\boldsymbol{G} = (G_1, \ldots, G_n)$ with $G_i = (V_i, E_i, L_{\mathrm{v}i}, L_{\mathrm{e}i}, l_i, r_i)$ $(1 \leq i \leq n)$, and $\mathcal{I} = (D, D_{\mathrm{i}}, D_{\mathrm{o}}, I)$, and let $j \in \{1, \ldots, n\}$ be such that G_j is the concurrent-Σ-algorithm main component graph. Then we can construct a non-deterministic sequential proto-algorithm $A' = (\Sigma', (G'), \mathcal{I}')$ from the concurrent proto-algorithm A such that $A \equiv_{\mathrm{a}} A'$ as described below.

Σ' is constructed from Σ as follows: $\Sigma' = (F', \emptyset, \emptyset, P')$, where:

$$F' = \{f_i \mid f \in \widetilde{F} \cup F_{\mathrm{s}} \cup F_{\mathrm{g}} \wedge i \in \{1, \ldots, n\}\} \cup \{\mathsf{ini}, \mathsf{fin}\};$$

$$P' = \{p_i \mid p \in P \wedge i \in \{1, \ldots, n\}\}.$$

$\mathcal{I}'$ is constructed from $\mathcal{I}$ as follows: $\mathcal{I}' = (D^{n+1}, D_{\mathrm{i}}, D_{\mathrm{o}}, I')$, where:

$I'(f)(d_1,\ldots,d_{n+1}) = (d_1,\ldots,d_{j-1},I(f)(d_j),d_{j+1},\ldots,d_{n+1})$ if $f \in \{\mathsf{ini},\mathsf{fin}\}$;

$I'(f_i)(d_1,\ldots,d_{n+1}) = (d_1,\ldots,d_{i-1},I(f)(d_i),d_{i+1},\ldots,d_{n+1})$ if $f \in \widetilde{F}$;

$I'(f_i)(d_1,\ldots,d_{n+1}) = (d_1,\ldots,d_n,I(f)(d_i,d_{n+1}))$ if $f \in F_{\mathrm{s}}$;

$I'(f_i)(d_1,\ldots,d_{n+1}) = (d_1,\ldots,d_{i-1},I(f)(d_i,d_{n+1}),d_{i+1},\ldots,d_{n+1})$ if $f \in F_{\mathrm{g}}$;

$I'(p_i)(d_1,\ldots,d_{n+1}) = I(p)(d_i)$ if $p \in P$.

G' is the restriction of a rooted labeled directed graph G to the vertices and edges reachable from its root. The graph G is constructed from $\boldsymbol{G}$ as follows: $G = (V,E,L_{\mathrm{v}},L_{\mathrm{e}},l,r)$, where:

$V = (V_1 \times \ldots \times V_n) \times \{1,\ldots,n\}$;

E is such that:

$(((v_1,\ldots,v_n),i),((v_1,\ldots,v_{i-1},v',v_{i+1},\ldots,v_n),i')) \in E$
if $(v_i,v') \in E_i$, $l(v_i) \in F \cup F_{\mathrm{s}} \cup F_{\mathrm{g}}$, $i' \in \{1,\ldots,n\}$, and $\mathrm{outdeg}(v_{i'}) > 0$;

$(((v_1,\ldots,v_n),i),((v_1,\ldots,v_{i-1},v',v_{i+1},\ldots,v_n),i)) \in E$
if $(v_i,v') \in E_i$ and $l(v_i) \in P$;

$L_{\mathrm{v}} = F' \cup P'$;

$L_{\mathrm{e}} = \{0,1\}$;

l is such that

$l(((v_1,\ldots,v_n),i)) = f$ if $l_i(v_i) = f$ and $f \in \{\mathsf{ini},\mathsf{fin}\}$;
$l(((v_1,\ldots,v_n),i)) = f_i$ if $l_i(v_i) = f$ and $f \in \widetilde{F} \cup F_{\mathrm{s}} \cup F_{\mathrm{g}}$;
$l(((v_1,\ldots,v_n),i)) = p_i$ if $l_i(v_i) = p$ and $p \in P$;
$l((((v_1,\ldots,v_n),i),((v'_1,\ldots,v'_n),i'))) = l_i((v_i,v'_i))$ if $l_i(v_i) \in P$;

$r = ((r_1,\ldots,r_n),j)$.

The non-deterministic sequential proto-algorithm A' is constructed in such a way that there exists a bijection from $\mathcal{S}_A$ to $\mathcal{S}_{A'}$ and that this bijection, say β, is such that, for all $s,s' \in \mathcal{S}_A$, $s \in \delta^{\mathrm{a}}_A(s')$ iff $\beta(s) \in \delta^{\mathrm{a}}_{A'}(\beta(s'))$. From

this, it follows immediately that the relations $R = \{(s, \beta(s)) \mid s \in \mathcal{S}_A\}$ and $R' = \{(\beta(s), s) \mid s \in \mathcal{S}_A\}$ witness algorithmic equivalence of A and A'. $\square$

Because concurrent algorithms are expected to be equivalence classes of concurrent proto-algorithms under algorithmic equivalence, Theorem 25 suggests that each concurrent proto-algorithm represents an algorithm that can as well be represented by a non-deterministic sequential proto-algorithm.

8 Concluding Remarks

Based on the classical informal notion of an algorithm, the notion of a classical proto-algorithm has been introduced in [9]. The notion of a concurrent proto-algorithm introduced in this paper is a generalization of that notion. Other interesting generalizations of the notion of a classical proto-algorithm are the notion of an interactive proto-algorithm and notions of a parallel proto-algorithm.

The notion of an interactive proto-algorithm has been introduced in [10]. To also cover (modulo algorithmic equivalence) the algorithms that are usually considered both concurrent and interactive, we can try to combine the notion of a concurrent proto-algorithm and the notion of an interactive proto-algorithm. It is to be expected that the resulting notion will be rather complex. Theorem 25 and the fact that the notion of an interactive proto-algorithm from [10] admits non-determinism suggest that we can instead cover those algorithms by the notion of an interactive proto-algorithm.

A notion of a parallel proto-algorithm is more concrete than the notion of a concurrent proto-algorithm. This means that the algorithms covered by a notion of a parallel proto-algorithm are also covered by the notion of a concurrent proto-algorithm. A notion of a parallel proto-algorithm may, for example, be useful in studying the time efficiency of concurrent algorithms implemented using a particular kind of parallelism.

Acknowledgement

I thank an anonymous referee for carefully reading a preliminary version of this paper and for pointing out some minor errors in it.

References

[1] Richard Cole and Ofer Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '89, page 169–178, New York, USA, 1989. Association for Computing Machinery. doi:10.1145/72935.72954.

[2] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, page 114–118, New York, USA, 1978. Association for Computing Machinery. doi:10.1145/800133.804339.

[3] Leslie M. Goldschlager. A universal interconnection pattern for parallel computers. *Journal of the ACM*, 29(4):1073–1086, 1982. doi:10.1145/322344.322353.

[4] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, second edition, 1990.

[5] Stephen Cole Kleene. *Mathematical Logic*. John Wiley and Sons, New York, 1967.

[6] Donald E. Knuth. *The Art of Computer Programming: The Fundamental Algorithms*. Addison Wesley Longman, Redwood City, CA, third edition, 1997.

[7] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71(1):95–132, 1990. doi:10.1016/0304-3975(90)90192-K.

[8] Anatoly Ivanovich Mal'cev. *Algorithm and Recursive Functions*. Wolters-Noordhoff, Groningen, NL, 1970.

[9] Cornelis A. Middelburg. On the formalization of the notion of an algorithm. In Ana Cavalcanti and James Baxter, editors, *The Practice of Formal Methods: Essays in Honour of Cliff Jones, Part II*, volume 14781 of *Lecture Notes in Computer Science*, pages 23–44. Springer-Verlag, 2024. doi:10.1007/978-3-031-66673-5_2.

[10] Cornelis A. Middelburg. Formalizing the notions of non-interactive and interactive algorithms. *Scientific Annals of Computer Science*, 35(2):211–249, 2025. doi:10.47743/SACS.2025.2.211.

[11] Naomi Nishimura. A model for asynchronous shared memory parallel computation. *SIAM Journal of Computing*, 23(6):1122–1147, 1994. doi:10.1137/S0097539791219670.

[12] Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.

[13] Larry Stockmeyer and Uzi Vishkin. Simulation of parallel random access machines by circuits. *SIAM Journal of Computing*, 13(2):409–422, 1984. doi:10.1137/0213027.

[14] Jiří Wiedermann. Parallel Turing machines. Technical Report RUU-CS-84-11, Department of Computer Science, University of Utrecht, 1984.

© **Scientific Annals of Computer Science 2026**